

GaliLEO User's Manual

V 0.2

Laurent Franck - Laurent.Franck@enst.fr
Francesco Potortì- F.Potorti@cnuce.cnr.it

DRAFT

Contents

1	Introduction	5
1.1	The Genesis	5
1.1.1	Ancestors and relatives	6
1.2	Scope and assumptions	6
1.3	Organisation of this document	6
1.4	Credits	7
2	The Framework	9
2.1	The philosophy	9
2.2	First layer: the simulation engine	10
2.2.1	Components as building blocks	10
2.2.2	Communication among components	10
2.3	Second layer: the customisable components	11
2.3.1	The source segment	12
2.3.2	The ground segment	13
2.3.3	The space segment	14
2.3.4	The simple life of a connection	16
2.4	Third layer: the customised components	19
3	Using GaliLEO	21
3.1	Compiling	21
3.2	Configuring	21
3.2.1	The simulation segment file	22
3.2.2	The space segment file	25
3.2.3	The ground segment file	29
3.2.4	The source segment file	33
3.3	Running	33
3.4	Getting the results	33
3.5	A sample experiment	34
3.5.1	The constellation	34
3.5.2	The traffic model	34

3.5.3	The stations	34
3.5.4	The satellite on-board processing	34
4	The Simulation Engine	35
4.1	Discrete event simulation	35
4.2	It's all about components	35
4.2.1	Code components	35
4.2.2	Customisable code components	35
4.2.3	Data components	35
4.2.4	Customisable code components	35
4.3	Being creative	35
4.3.1	Creating a new agenda	35
4.3.2	Creating a new scheduler	35
4.3.3	Creating new components	35
5	The Components	37
6	Examples of customisation	39
6.1	Creating new traffic sources	39
6.2	Creating a new ISL routing algorithm	39
6.3	Creating a new link state manager	39
6.4	Creating a new connection change manager	39
6.5	Creating new measures	39

Chapter 1

Introduction

1.1 The Genesis

Papers found in the networking literature usually start with a sentence such as: “Telecommunication networks feature an ever growing complexity”, which, while obvious, is the motivation behind the same existence of GaliLEO. In fact, while evaluating the performance of a network of communication satellites necessarily requires the use of appropriate simulation tools, there are few such tools. Some commercial well known tools, such as Opnet [?], which also sports a specialised satellite module, or Bones [?], are expensive and have broad scope. Publicly available tools such as Ptolemy [?] also have a broader scope than what is needed, and Ns [?], on the contrary, supports now constellations, but is strongly oriented towards packet level simulation in IP environments.

The idea of GaliLEO germinated among some members of the Cost 253 Action *Efficient LAN interconnection through constellation of satellites* [?] funded by the European Commission. Cost 253 is not aimed at building such a simulation tool, however we were the witnesses of this lack of tools which would be needed for conducting the research feeding the Cost 253 activities. As a result, a collaboration was initiated involving three institutions:

- The Computer Network group of CNUCE in Pisa, Italy. CNUCE institute is part of the CNR, the Italian National Research Council.
- The Ecole Nationale Supérieure des Télécommunications (ENST) through its site of Toulouse. ENST is an engineering high school whose Toulouse branch is specialised in satellite communications.
- The Public University of Navarra, Spain.

A fourth institute, namely CSELT (Telecom Italia Research Centre) joined the effort when it was foreseen that GaliLEO could be interfaced with CONSIM, a

simulator developed in CSELT aimed at evaluating the impact of failures in constellation environments [?].

The resulting work in progress was baptised GaliLEO, and can be concisely described as “A simulation framework for studying communication networks issues in satellites constellations”. The networking aspects covered at present are satellite access techniques, routing in ISL network (including signalling), and handover management. GaliLEO is not focused on a given technology which means that is not possible to do detailed, low-level simulations of a given protocol or transmission medium. On the other hand, the versatility of GaliLEO should make it possible to simulate very big and complex system, by putting the emphasis where it is needed. Also, provisions can be made to have it interact with technology-specific tools, such as CONSIM, thus treating it as a high level tool for macroscopic evaluations which can be refined later using tools specialised for a given technology or a given transmission aspect.

In order to achieve these goals, GaliLEO makes extensive use of discrete event simulation and object oriented design. Last but not least, GaliLEO will be free software [?], distributed under the GNU Public License [?]. It will also be free of charge for everyone.

1.1.1 Ancestors and relatives

GaliLEO was not born out of the blue, but has some history behind it. Also, it plans not stay alone, but to interact with other simulation tools.

Fracas

SimToc

LeoSim

CONSIM

1.2 Scope and assumptions

1.3 Organisation of this document

Chapter 2 provides a description of how GaliLEO is structured. A special emphasis is put on the network model and the interactions among components. Chapter 3 gives all information needed to run GaliLEO: compilation, the configuration files, how to run it and how to gather the results. Chapter 4 goes into the gory details of the simulation engine. It lays the foundations for Chapter 5 which is

a reference guide of all components involved in customising GaliLEO. Finally, chapter 6 provides examples of GaliLEO customisations.

If you intend to use GaliLEO without creating new components then reading Chapters 2 and 3 will be sufficient. The ambitious user of GaliLEO will go through all chapters.

1.4 Credits

GaliLEO is the product of cooperation between people thinking that it would be nice to have a simulation framework freely available for everybody. Here is the hall of fame:

Marco Annoni (*marco.annoni@cselt.it*): fault management, unconditional support.

Nedo Celandroni (*nedo.celandroni@cnuce.cnr.it*): administrative management, design, hosting.

Erina Ferro (*erina.ferro@cnuce.cnr.it*): administrative management, design, hosting.

Laurent Franck (*laurent.franck@ens.fr*): design, coding, documentation, web.

Mikel Izal (*mikel.izal@unavarra.es*): design, coding.

Francesco Potorti (*f.potorti@cnuce.cnr.it*): design, coding, documentation, make-file wizardry.

It must also be noted that without the Cost253 action and therefore the EC funding this project would not have commenced.

Our secret wish is that, within 1 year, the credits list will be permanently obsoleted and a pain to maintain [today is 01/06/00].

DRAFT

Chapter 2

The Framework

This chapter provides a description of how GaliLEO is organised. It introduces the most important concept in GaliLEO: the component (available in different flavours to suit your individual taste). This chapter also presents the network model which is implemented in GaliLEO, hence drawing a clear boundary between what is within the GaliLEO scope and what is not.

The next section goes briefly into what is called the philosophy of GaliLEO which is articulated around a three layer architecture. The remaining sections detail the contents of the three layers, discussing the where, what and why.

2.1 The philosophy

GaliLEO aims at providing a simulation framework where one plugs his home brewed routing algorithm, or traffic generator, or priority call dropping policy, or whatever, and evaluates the resulting behaviour. In a first sketch, GaliLEO can be split in two parts: the simulation core and the customised components. In order to develop the customised components two approaches were considered: (a) to use a specific language for interfacing with the simulation core or (b) to directly interface with the simulation core as with a huge library accessed through a well defined API. The first approach is more complex but is also cleaner. The second approach, which is the one taken in GaliLEO, is more flexible but the learning curve is certainly steeper.

GaliLEO is written in Java, so implementing customised components boils down to writing new Java classes deriving from existing classes found in the simulation core, and communicating using methods from the simulation core classes. The whole architecture has a three layer organisation. The first layer introduces the concept of components which are GaliLEO building blocks. This layer also provides the communication facility using a discrete event paradigm. The second

layer implements the network model of GaliLEO. The network model defines the scope of GaliLEO possibilities using a set of classes. These classes also specify the rules for using the network model and creating customised components. Finally, the third layer is a nebula of customised components orbiting around the core made of the first and second layer.

2.2 First layer: the simulation engine

This layer introduces two mechanisms. The first one permits to create building blocks called components. The second one provides a mean to exchange messages among components. These two mechanisms are used extensively in the upper layers.

2.2.1 Components as building blocks

Components are the building blocks of GaliLEO. There are two categories of components: *code components* and *data components*. Code component have a processing vocation while data components are meant to store data.

Code components are further divided into regular code components (which as of now are referred to as *code components*), *customisable code components* and *entities*. Customisable code components extend code components by adding the possibility to be initialised from a configuration file at run time. Entities are code components able to receive events, this topic is covered below.

Data components also exist in two flavours: regular *data components* and *customisable data components*. These latter can be initialised from a configuration file at run time.

All components provide the ability to be duplicated and a method to initialise the component after creation. Code component have an additional service to start the operation of the component. It should be noted that only entities can be seen as independent pieces of code that run concurrently. The GaliLEO network model described later is therefore a collection of inter operating components.

2.2.2 Communication among components

Communication among components corresponds to invoking the methods of other components and is implemented using an event passing scheme. This type of method invocation is different from what is existing in common object oriented languages, since the call for the method can be asynchronous, i.e. scheduled to happen in the future. This whole communication facility boils down to the inclusion of a simulation engine with events, events scheduler and processing nodes.

An event dispatches a *selector* in the GaliLEO jargon. For each component method that must be callable from another component, a selector is made available that implements the actual method call. When posting an event to the event scheduler, the calling component specifies to which component and which selector the event is directed. The method is implicitly identified by the type of the selector. Checking is done to ensure that the callee component accepts that type of selector, i.e. that it includes the method corresponding to the selector. Furthermore an event can have parameters attached, these parameters are used by the selector and then passed to the method call. Figure 2.1 summarises the relation between components and selectors.

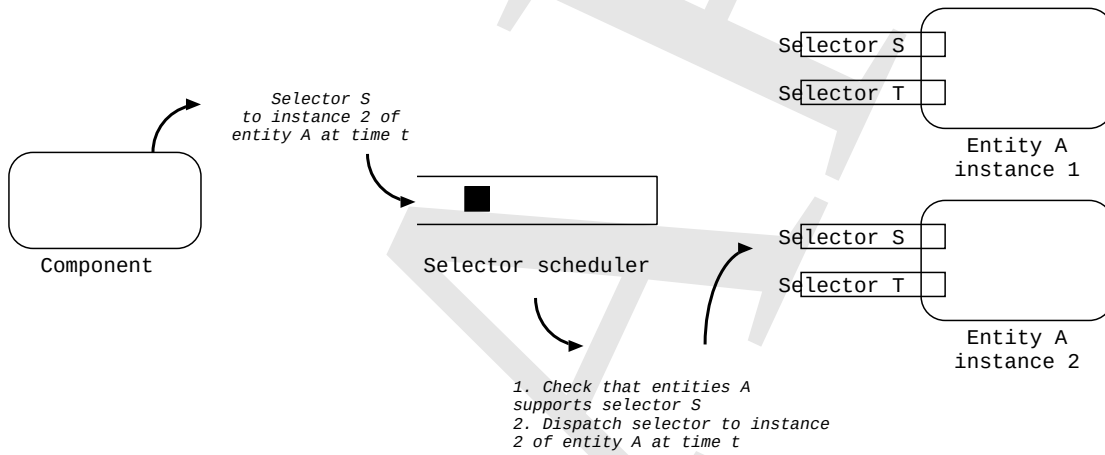


Figure 2.1: Relation among components and selectors.

2.3 Second layer: the customisable components

using the services offered by the first layer, it is possible to write classes implementing new components and to have components communicating among them. These facilities are used to create a set of classes implementing the GaliLEO network model. In addition to these classes, other classes are also created in order to assist the user in the simulator customisation. These additional classes provide tools to collect measurements or for manipulating data structures such as a graphs, linked list etcetera. Section ?? will focus on the presentation of the network model, while Chapter 5 discusses the internal of the customisable classes and the additional tools.

The network model provides all elements to simulate a constellation of satellites and the ground stations that access it. Component pertain to one of the main

network *segments*: the source segment, the ground segment, and the space segment. The source segment deals with traffic generation, the ground segment deals with the ground stations and the space segment deals with everything related to the satellites.

2.3.1 The source segment

The source segment includes all components responsible for generating traffic. Its structure is represented in figure 2.2.

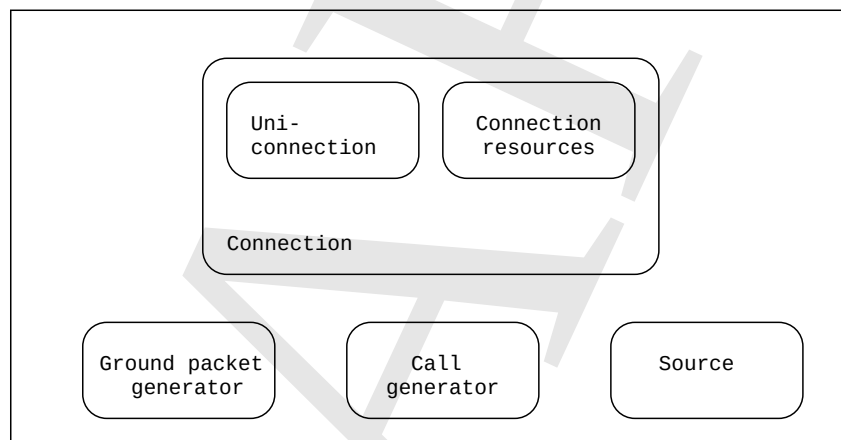


Figure 2.2: The source segment.

The corner stone of the source segment is the call generator. Each generator represents a source of traffic. A call generator provides answers to the following questions (among others):

- When will the first connection request (for this call generator) be issued.
- What are the characteristics (e.g. source, destination and requirements) of a connection request (a connection is formed of two unidirectional connections).
- When will the requirements for the connection be modified (tear down is consider as a special case of modification).
- How will the requirements of the connection be modified.
- When is the next connection request to be issued.

The connection requirements are expressed in terms of connection resources, of which many types may coexist. Call generators are of two types: regular and constellation restricted. The former connects two stations through two unidirectional connections. The latter connects two satellites. The purpose of constellation restricted connections is to load the constellation with background traffic.

For the moment being, simulation is performed at the connection level, however it is expected that packet level simulation will be implemented in GaliLEO. For this reason, a ground packet generator can be attached to each call generator. This feature is not yet implemented.

The source component is responsible for querying the call generators about connection requests, and for transmitting these requests to the call signalling of the appropriate stations or satellites. The same applies for modifications of existing connections. Subsection 2.3.4 gives a complete trace of a connection lifetime.

2.3.2 The ground segment

The source segment implements everything related to the station. Figure 2.3 represents the structure of one station (there may be many).

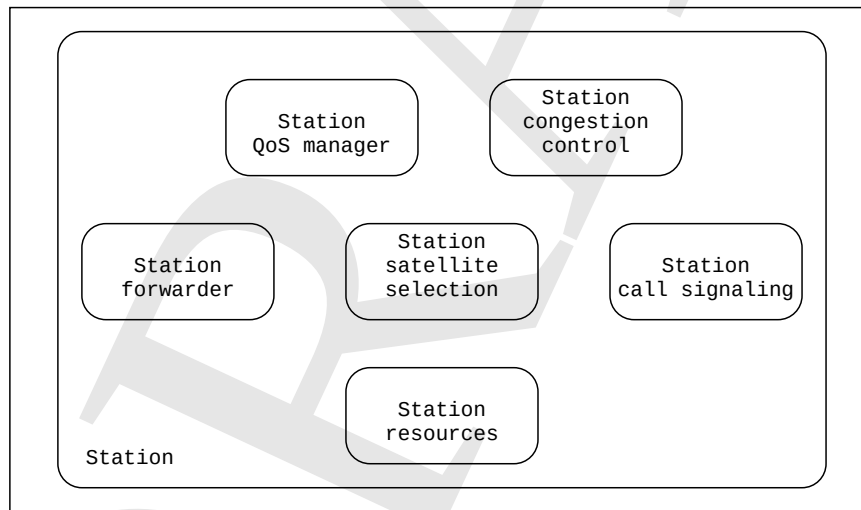


Figure 2.3: Ground segment: the structure of a station.

The station component includes the other components showed in the figure. The connection resources component models the resources that are present in the station (resources can represent the available capacity for example). The resources status evolves as connections are setup and torn down. The QoS manager handles all operations on the resources by providing allocation, deallocation of resources

as well as call admission control. During the connection setup, the satellite selection component selects the first and last hop satellites.

Provision for packet forwarding is already included with a forwarder component, whose role is to adequately dispatch the packets to the up link, and a congestion control component that monitors the traffic and takes the appropriate actions (such as throttling down the source) if congestion takes place.

2.3.3 The space segment

The space segment is the most complex segment; it encompasses aspects related to both the constellation as a whole, and the individual satellites.

The constellation

Figure 2.4 represents the components in the constellation part of the space segment.

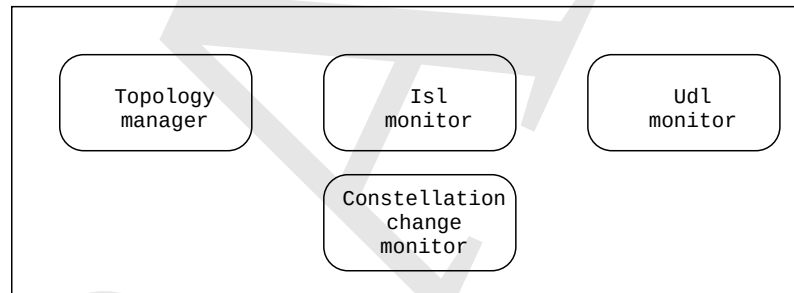


Figure 2.4: Space segment: constellation aspects.

The topology manager component implements all the orbital mechanics. It is able to determine:

- What is the position of a satellite according to time.
- What is the state (*enabled*, *disabled*) of an ISL (Inter-satellite link) according to time.
- What is the delay before an ISL switches its state.
- What is the elevation angle of a satellite with respect to a station, according to time.
- What is the delay before a satellite passes below the horizon of a station according to time.

Each type of constellation (polar, inclined, ...) requires a customised implementation of the topology manager.

The ISL monitor keeps a watchful eye on the evolution of the state of the ISLs in the constellation. When an ISL switches its state, all other components that need to be warned about this event are notified. This mechanism is used, for example, in order to manage handovers.

The UDL monitor does the same task for the UDLs. When a station leaves a satellite footprint, the UDL monitor notifies all involved actors in order to ensure a smooth UDL handover.

The constellation change monitor takes care of managing handovers when a link (UDL or ISL) is switched off. The connection change manager uses the services of the ISL and UDL monitors to be notified when such an even happen.

The satellite

Figure 2.5 represents all components that are part of the internal structure of the satellite.

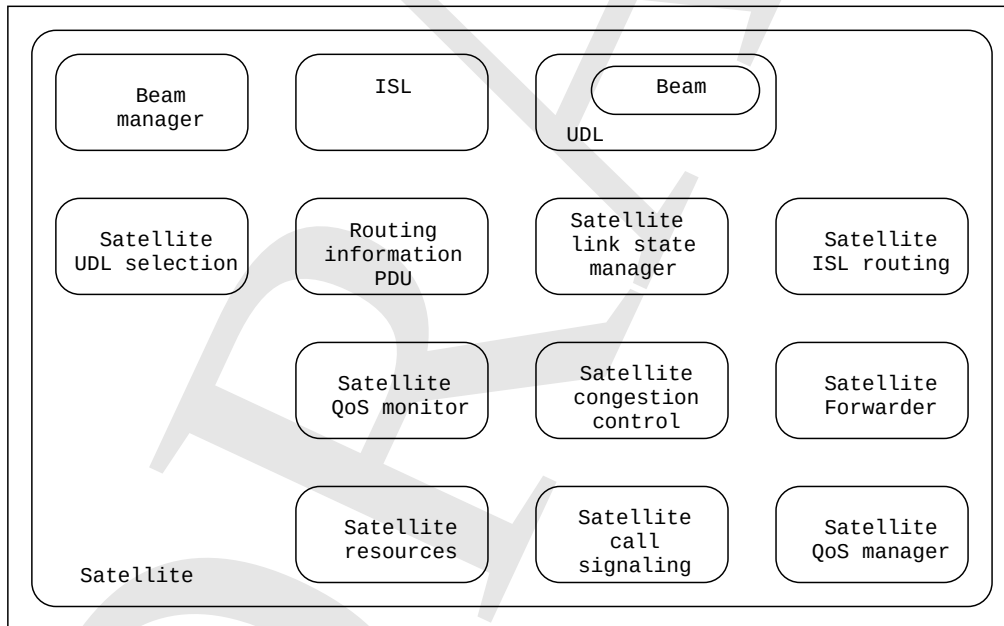


Figure 2.5: Space segment: the structure of a satellite.

Similarly to a station, a satellite has a resources component describing the resources in the satellite. The satellite includes a call signalling component to process requests for connection setup, modification and tear-down. The dynamics of

the connections have an impact on the satellite resources status and the QoS manager allocates, deallocates resources and performs call admission control. During the call setup phase, the UDL selection component is invoked in order to find out which UDL will support the connection. Once the UDL is selected, the beam manager component is in turn invoked to decide about which beam inside the selected UDL will be assigned to the connection.

In order to support packet level simulation, a forwarder, congestion control and QoS monitor components are also present but currently disabled.

The ISL routing component is responsible for computing routes within the constellation between a first and last hop satellite. In order to compute accurate routes, it needs routing information reflecting the network state. This routing information is gathered (and distributed) by the link state manager component in the form of routing information PDUs (Protocol Data Unit).

All satellites in the constellation have the same structure, it is currently impossible to create hybrid constellations where satellites have different payloads. ****It shouldn't be too difficult, indeed, and we should really implement it****.

2.3.4 The simple life of a connection

This subsection provides a trace of the action taking place when a connection is set up, modified and eventually torn down. Only connections from station to station are covered. The constellation restricted case, i.e. from satellite to satellite, is a special case where all aspects related to the stations are omitted. Constellation restricted connections are only set up and then torn down, no modification occurring in between.

Connection creation and setup

**** Ditto for connection re-routing which can be seen as a setting up a new connection****. ****More or less careful reading and correction by Francesco up to here, 20000802****

****The following should be more detailed, and start from when the Source initialises the generators**** The connection is created by the call generator which controls the characteristics of the connection such as: source and destination stations and resources requested. When queried for, the call generator returns a connection component made of two uniconnections.

Once the connection is created, it is passed from the source component to the call signalling of the source station. The call setup is a two phase process. During the first phase, satellite selection, ISL routing, UDL and beam selection and call admission control are performed. During the second phase the connection

is established and resources are allocated. These two phases are conducted for each unconnection forming the connection.

The first phase can be detailed as follows. In the call signalling component of the source station, the satellite selection of the station is invoked to determine what are the first and last hop satellites. Then the QoS manager of the source station is called to perform call admission control. The call admission control can decrease the amount requested resources. If the unconnection is granted admission, the setup request is forwarded to the first satellite. This satellite performs ISL routing to determine the route between the first and last hop satellites. It also performs UDL selection and beam selection. If these operations succeed, call admission is invoked and the request is forwarded to the next satellite (provided that enough resources are available). Each satellite on the route performs call admission control. The last satellite performs UDL selection, beam selection and then forward the request to the destination station which, in turn, will perform call admission control. This ends the first phase.

The second phase consists in forwarding an allocation request starting from the source station to the first satellite until the destination station. Because the call admission control has preallocated the resources, the allocation phase cannot fail.

The two phase signalling procedure is implemented in GaliLEO with regular method calls, i.e. not using events. The choice is motivated by the requirement to avoid the overhead of event creation and scheduling. This concern is relevant since the call signalling is likely to be often invoked. On the other hand, that approach has the disadvantage of making it impossible to model delay during each hop between nodes. An hybrid solution was adopted where, during each hop, a variable accounts for the delay spent in each node. Then an event is used to carry the confirmation of the connection setup to the source. This event is delayed according to the delay accounted.

After the source component receives the confirmation for the connection setup, it invokes the call generator to know about:

- When the next connection setup request must be issued. A special value is used to mean 'never'.
- When the next modification request for this connection must be issued.

Based on the information provided by the call generator, the source component scheduled for itself two events (setup and modify) according to the delay returned by the call generator. When the event is eventually triggered, the next procedure described hereafter is performed.

Connection modification

The source component first queries the call generator to learn about the modification which is requested. As currently implemented, the call generator either increases or decreases the amount of resources requested by the connection. Then the modification request is passed to the call signalling of the station. The modification is handled the same way a connection request is processed using a two phase process. However the following differences exist:

- The connection is not rerouted, therefore neither the satellite selection nor the ISL routing are invoked.
- If the modification requested implies a decrease of the connection resources, the first phase (call admission control) never fails.

Once the signalling is completed, the source component is notified of its completion. The call generator is then queried to know when the next modification will take place (provided that the modification succeeded) and a new modification event is scheduled.

Connection rerouting

Because the satellites are moving with respect to the earth stations and because some constellations have non permanent ISLs, a connection may require rerouting because one of its links is broken. The connection might also be rerouting as a result of a less critical event such as for example a significant length increase of one of its links. The Udl and Isl monitors keep track of the evolution of the link states and when something relevant happens, the connection change monitor is invoked. The connection change monitor will, in turn, perform the appropriate action which is, commonly, rerouting the connection.

Connection tear-down

Connection tear-down is a special case of connection modification. When the call generator is queried for the type of modification, a special value is returned meaning connection tear-down. Connection tear-down also happens as the result of failure in modifying a connection.

Once the connection tear-down confirmation has reached the source, the call generator is queried about when the next connection setup request must be scheduled. Referring back to the description of the connection setup, it turns out there are two opportunities for scheduling connection requests:

1. After a connection is setup.

2. After a connection is torn down.

By using one scheme or the other, is possible to have either concurrent or interleaved connections. It is also possible to have both schemes at the same time.

2.4 Third layer: the customised components

The previous section described the network model implemented in GaliLEO. The purpose of GaliLEO is to implement different schemes (e.g. routing algorithms) and compare how they behave. As a result, one must be offered the opportunity to customise the ISL routing component (or any other). It must even be possible to create a new component in order to replace an existing one. During this customisation/create process, some issues must be addressed:

- The customised component must fulfil the role it is intended to, say for example, computing routes.
- The customised component must fit in the framework and be able to inter-operate with the other components.

The customisation process must therefore be controlled while not giving away too much flexibility. It was decided that, when creating a new component, this component have always to be based (directly or transitively) on a component which is part of the simulation core. This initial component, called the customisable component, defines all rules that must be fulfilled by any customised component claiming affiliation to the customisable component. To summarise:

1. Customisable components ensure that the customised components fit in the framework.
2. Only customised components deriving from customisable components can be plugged in the simulation core.

As a result, almost all components part of the network model are customisable components and are meant to be extended to build customised components. As a matter of fact, these customisable components have such a basic form that they are not working as such. The customisable components specify what are the services that must be provided (e.g. a service for computing routes in the case of ISL routing algorithms). That policy helps ensuring the concern that a customised component will do its duty.

The second concern, that is, interoperability with other components, is tackled in two ways:

- All customised component have to provide a set of well defined services common to all components. For example, each component must provide a *duplicate* service which makes a perfect clone of the component.
- Depending on the component type (say ISL routing) services must be provided in order to check that the customised ISL routing is able to cooperate with other customised components. For example, a customised ISL routing must check whether it is able to handle the customised connection resources that are used for the simulation.

At the beginning of the simulation, after all customised components are loaded in the system, integrity checks are conducted to ensure that all customised components are able to cooperate and that they all provide the basic services mentioned previously (such as the *duplicate* service).

GaliLEO aims at being as flexible as possible. It is therefore not surprising that almost all component of the network model are customisable. However, there are some exceptions listed below:

The source component : the way the source component behaves cannot be modified. However, using different call generators, a fairly high level of flexibility (well, in our opinion) can be achieved.

The connection component : connections (as well as unconnections) cannot be customised, although connection resources can.

The station component : the structure of station is fixed, it is impossible to add, for example, a network management component. On the other hand, most of the components that are in the station are customisable.

The station call signalling component : the call signalling is the exception to the above statement since it is not customisable. We believe that the need for evaluating different call signalling protocols is (at least currently) not within the scope of GaliLEO.

The satellite component : for the same motivation that made the station a fixed component, the satellite is not customisable.

The satellite call signalling component : obviously the justification is similar to the one for the station call signalling.

All the other components presented in Section 2.3 are customisable. Some components present in the simulation segment are also customisable, this point is addressed in Chapter 4. Chapters 5 and 6 dive in the gory details of customising components.

Chapter 3

Using GaliLEO

This chapter deals with the usage of GaliLEO. How to compile it, configure it, run it and ultimately, how to fetch the results. The chapter ends with a description of a sample configuration called *experiment*.

3.1 Compiling

GaliLEO was successfully compiled on Linux systems using the following compilers:

- Blackdown's port of Sun's JDK 1.2.2 RC 4: a standard implementation of Java 2. Now available from Sun.
- IBM's JDK 1.1.8: a faaaast virtual machine. Meep-meep.
- GNU gcj 2.95.2: which is the version provided with Suse 6.4.

The tarball contains a `GaliLEO` directory storing everything. In this directory, a Makefile takes care of compilation. `make` creates the class files using the selected JDK (either explicitly specified in the Makefile or using the `JAVA` and `JAVAC` environment variables or by looking in the path). `make allgcj` operates similarly using gcj instead of a JDK. All source files are compiled in `.o` and then linked into an executable surprisingly called `galileo`.

3.2 Configuring

The configuration is controlled through four files:

The simulation segment file : providing the parameters for configuring the simulation engine and configure the component used for logging the results.

The space segment file : providing the parameters for defining the structure of the constellation and dimensioning the space segment.

The ground segment file : providing the parameters for defining the structure of the ground stations, dimensioning the stations and positioning the stations on the earth surface.

The source segment file : for specifying the type and parameters of the traffic sources.

These configuration files use respectively the suffixes `.sim`, `.spa`, `.gro` and `.sou` for naming. The configuration syntax is made of nested blocks. A block starts with a keyword specifying what is configured (e.g. `satellite`) and then an opening curly brace. The block ends with a closing curly brace.

For the configuration of customised component the structure is the following. After the opening curly brace, the `Component` keyword is used followed by the name (including the package name) of a Java class. If the customised package can be parametrized then a list of (keyword,value) pairs follows. The following example defines a customised topology manager for polar constellations.

```
TopologyManager {
    Component GaliLEO.Library.Constellation.PolarConstellation
    NumberOfOrbits 6
    SatsPerOrbit 11
    OrbitAltitude 780
    SatellitePhase 16.36
    OrbitDelta 31.6
    OrbitInclination 86.4
    MinimumElevation 8.2
    IslCutoffAngle 30
}
```

3.2.1 The simulation segment file

Figure 3.1 shows a sample configuration file for the simulation segment.

Configuration of the debugging traces

GaliLEO can be configured to produce a substantial amount of debugging information. Debugging information verbosity is controlled through two parameters:

DebuggingType : what parts of GaliLEO will produce debugging information. The class `GaliLEO.Debug.Debug.java` contains a list of codes for the different parts.

```
// A simple configuration file: simple.sim

DebuggingSupport {

    DebuggingType 255
    DebuggingLevel 3

}

SimulationEngine {

    Scheduler {
        Component GaliLEO.Library.Engine.SimpleScheduler
    }

    Agenda {
        Component GaliLEO.Library.Engine.Calendar
        CalendarSize 100
        SlotSize 10
    }

    Logger {
        MessageOutput display
        MeasureOutput display
        LoggingPeriod 100
    }

}

PeriodicalMeasures {
    ConnectionRequests call_signaling false
    BlockedConnections call_signaling false
    NumberOfRoutingInformationBroadcasts routing_signaling true
}
```

Figure 3.1: A sample configuration file for the simulation segment.

`DebuggingLevel` : how verbose the information is allowed to be. The higher the level, the more verbose the debugging information is.

Note: in order to enable the output of debugging messages, a special variable (`Debug.Debug.debug_mode`) must be set to `true` and the class must be recompiled.

Configuration of the simulation engine

The engine is configured by specifying:

- Which event `Scheduler` is used. In the sample configuration file, a simple scheduler is used providing sequential scheduling of events. `SimpleScheduler` is the name (including the package identification) of the class implementing the scheduler.
- Which `Agenda` is used for queueing the scheduled events. The agenda used here (`Calendar`) is parametrised using two values: `CalendarSize` and `SlotSize`. The former specifies the number of slots in the calendar while the latter specifies the time span of a slot. Another type of agenda is available: the `DeltaList`. Nevertheless, it is less efficient and is available for historical reasons (yes, GaliLEO has an history, see the introduction chapter !!!).
- Where the `Logger` must write the results. Three parameters are provided. `MessageOutput` which controls where messages are displayed. The `display` keyword means the standard console while any other string indicates a filename. `MeasureOutput` is similar. `display` routes the output of the measures to the console. `file` tells GaliLEO to output the measures on file. No filename is specified here since one file is used per measure, the filenames used for logging are constructed according to the measure name. Finally, `LoggingPeriod` specifies at which interval measures are logged. The time is expressed in seconds of simulated time. A value of 0 disables logging.

Configuration of the measure output

It was mentioned that the logger dumps on file or on the console the values of the measures which contain the results of the simulation. The set of measures that are dumped is described in this section. The sample configuration file shows three measures described each by three parameters.

The first parameter is the measure name. It must identify a measure known in the simulation, i.e. a measure implemented by one of the components. GaliLEO

will complain if it is not able to find the corresponding measure. The second parameter is a string describing the category of the measure. It is currently unused. The third parameter is a boolean indicating whether the value of the measure must be reinitialized after being logged.

3.2.2 The space segment file

The configuration file for the space segment file specifies everything about the satellite and the constellation. Figures 3.2 and 3.3 provide a sample space segment configuration file.

The configuration file is split in two parts. The first part specifies components global to the constellation. The second part describes the structure of the satellite.

The constellation

The description of the constellation relies on three components: `TopologyManager`, `UdlMonitor` and `IslMonitor`.

- The `TopologyManager` is the component implementing the orbital mechanics of the constellation. In this example, a `PolarConstellation` is used which implements a polar constellation with a seam. Another topology manager (not represented here) is the `PolarStaticConstellation`, a polar constellation with motionless satellites. This unrealistic constellation is useful for debugging purposes. Each topology manager is parametrised using a set of (keyword,value) pairs which are specific to the component.
- The `IslMonitor` block specifies which component is used for tracking the evolution of the ISLs in the constellation and notifying other components when a significant change happens. This component must be compatible with the ISL resources (see below).
- The `UdlMonitor` block specifies which component is used for tracking the evolution of the UDLs in the system. It must be compatible with the UDL resources (described below).
- The `ConnectionChangeMonitor` specifies which component is used to react to changes affecting the connections such as a ISL being switched off.

The satellite

The description of the satellite which is provided in the second part of the file is used to initialise all the satellites of the constellation. As a result, it is currently

```
// A simple space segment config file: simple.spa
```

```
TopologyManager {
  Component Galileo.Library.Constellation.PolarConstellation
  NumberOfOrbits 6
  SatsPerOrbit 11
  OrbitAltitude 780
  SatellitePhase 16.36
  OrbitDelta 31.6
  OrbitInclination 86.4
  MinimumElevation 8.2
  IslCutoffAngle 30
}

IslMonitor {
  Component Galileo.Library.Constellation.DummyIslMonitor
}

UdlMonitor {
  Component Galileo.Library.Constellation.DummyUdlMonitor
}

ConnectionChangeMonitor {
  Component Galileo.Experiment.Constellation.ConnectionChangeMonitor
}

Satellite {
  Resources {
    Component Galileo.Library.Satellite.SimpleSatelliteResources
    InitialResourcesAmount 100
  }

  Forwarder {
    Component Galileo.Library.Satellite.DummySatelliteForwarder
  }

  QoSManager {
    Component Galileo.Library.Satellite.SatelliteSimpleQoSManager
  }

  QoSMonitor {
    Component Galileo.Library.Satellite.DummySatelliteQoSMonitor
  }

  CongestionControl {
    Component Galileo.Library.Satellite.DummySatelliteCongestionControl
  }
}
```

```
IslRouting {
  Component Galileo.Library.Satellite.SimpleIslRouting
}

UdlSelection {
  Component Galileo.Library.Satellite.DummySatelliteUdlSelection
}

LinkstateManager {
  RoutingInformationPDU Galileo.Library.Satellite.SimpleRoutingInformationPDU
  RoutingInformation Galileo.Library.Satellite.SimpleRoutingInformation
  Component Galileo.Library.Satellite.UnrealisticPeriodicalLsManager
  FirstBroadcast 5
  BroadcastPeriod 100
}
}

Isl {
  Resources {
    Component Galileo.Library.Isl.SimpleIslResources
    InitialResourcesAmount 100
  }
}

Udl {
  Resources {
    Component Galileo.Library.Udl.SimpleUdlResources
    InitialResourcesAmount 100
  }

  Beam {
    Resources {
      Component Galileo.Library.Udl.SimpleBeamResources
      InitialResourcesAmount 100
    }
  }

  BeamManager {
    Component Galileo.Library.Udl.DummyBeamManager
  }
}
```

Figure 3.3: A sample configuration file for the space segment (cont'd).

impossible to create hybrid constellations with different satellites. Each satellite is described by the characteristics of its on-board processing and its ISL, UDL and Beam.

The on-board processing is described by the components `Resources`, `Forwarder`, `QoSManager`, `QoSMonitor`, `CongestionControl`, `ConnectionChangeMonitor`, `IslRouting`, `UdlSelection` and `LinkstateManager`.

- The `Resources` block specifies the what type of satellite resources is used and provides initial values for dimensioning the satellite. In the example, the resources description is collapsed into a single integer which is initialised to a value of 100 units.
- The `Forwarder` block specifies the component used for packet forwarding. This feature is not yet supported.
- The `QoSManager` block specifies which component is used for managing the resources of the satellite. In this case a `SatelliteSimpleQoSManager` is provided. This QoS manager knows how to handle `SimpleSatelliteResources` and `SimpleConnectionResources` and manages the resources according to a firm allocation scheme.
- The `QoSMonitor` block specifies which component is used to perform monitoring of the Quality of Service on connections. This feature is not yet supported.
- The `CongestionControl` block specifies which component is used to perform congestion control on the connections. This feature is currently not activated and will probably be when packet level simulation is implemented.
- The `IslRouting` block specifies which component is used to compute the routes inside the constellation. The `SimpleIslRouting` component used here knows about `SimpleConnectionResources` as well as `SimpleRoutingInformation` and computes routes according to a Dijkstra shortest path. The metric used is the number of hops (which is not very appropriate in satellite constellations). Furthermore, all links and satellites having insufficient resources for supporting the requirement of the connection are not taken into account for the route computation.
- The `UdlSelection` block specifies the name of component performing UDL selection during the signaling phase of the connection setup. This component must know about connection resources and UDL resources.

- The `LinkstateManager` block specifies which component is used implement signaling of routing information. Before providing the name of the component two additional information must be specified: the type of the `RoutingInformationPDU` used and the type of `RoutingInformation` handled by the link state manager. The link state manager used in this example implements a scheme where each satellite broadcasts information about its available resources every 100 seconds. The first broadcast is issued at time 5 s. The routing information PDU broadcast by this link state manager contains the amount of satellite resources available in the satellite broadcasting the information as well as the amount of resources available on each of its ongoing ISLs.

A specification of the characteristics of the ISLs, UDLs, and Beams is then included using the `Isl`, `Udl` and `Beam` blocks.

- The `IslResources` keyword in the `Isl` block specifies what component is used for defining ISL resources. In this case, a simple resource definition is used whose structure is similar to `SimpleSatelliteResources`.
- The `Udl` definition is made of three parts. The first one defines the type of resources used in UDLs (`UdlResources`). The second part (the `Beam` block) defines the resources for the Beams (`BeamResources`). The last part defines the component used for beam allocation and selection (`BeamManager`).

3.2.3 The ground segment file

Figure 3.4 shows a sample configuration file for the ground segment. The file is divided in two parts: the definition of station types and the positioning of the stations on the earth surface. Defining station types is useful for having configuration where all stations do not have the same internal architecture. When stations are positioned on earth, a reference is made to a previously defined station type.

Definition of station type

This part of the file defines what is inside a station. It consists in specifying these blocks: `Resources`, `Forwarder`, `QoSManager`, `SatelliteSelection`, `CongestionControl`. Before specifying the components a name must be assigned to the station type using the `TypeName` keyword. In the sample configuration file the station type is called `SimpleStation`.

```
StationType {  
    TypeName SimpleStation  
  
    Resources {  
        Component Galileo.Library.Station.SimpleStationResources  
        InitialResourcesAmount 100  
    }  
  
    Forwarder {  
        Component Galileo.Library.Station.DummyStationForwarder  
    }  
  
    QoSManager {  
        Component Galileo.Library.Station.StationSimpleQoSManager  
    }  
  
    SatelliteSelection {  
        Component Galileo.Library.Station.SimpleSatelliteSelection  
    }  
  
    CongestionControl {  
        Component Galileo.Library.Station.DummyStationCongestionControl  
    }  
}  
  
Station {  
    Type SimpleStation  
    Longitude 0  
    Latitude 0  
}  
  
Station {  
    Type SimpleStation  
    Longitude 0  
    Latitude 70  
}
```

Figure 3.4: A sample configuration file for the ground segment.

- The `Resources` block specifies what type of component defines station resources. The `SimpleStationResources` used here are similar to the `SimpleSatelliteResources`.
- The `Forwarder` block specifies the component used for forwarding packets to the constellation. This feature is not yet supported.
- The `QoSManager` block specifies the component managing the resources in the station. It is similar to the QoS manager of the satellite.
- The `SatelliteSelection` block specifies which component is used for satellite selection. This component selects, according to a connection request what are the first and last hop satellites for the connection. In this example file, the `SimpleSatelliteSelection` selects the first satellite which is visible from the station.
- The `CongestionControl` block specifies which component performs congestion control. This feature is not implemented and will be when packet level simulation is considered.

After defining one or more station types, it is possible to instantiate stations and position them on the earth.

Positioning of stations

Each station is defined as follows:

- The `Type` keyword specifies the type of the station. It must refer to a previously station type definition.
- The `Longitude` keyword specifies the longitude of the station. It is comprised between -180° (westbound) and 180° (eastbound).
- The `Latitude` keyword specifies the latitude of the station. It is comprised between -90° (southward) and 90° (northward).

When created, stations are numbered starting from 0. This identifier is used by the call generator (see below) to specify, when creating a connection, what are the source and destination stations.

```
Source {  
    CallGenerator {  
        Resources Galileo.Library.Connection.SimpleConnectionResources  
        Component Galileo.Library.Source.DeterministicCallGenerator  
        FirstConnectionDelay 10  
        InterArrivalDelay 20  
        ConnectionDuration 180  
        ConstellationRestricted false  
        Source 0  
        Destination 1  
        ResourcesAmountRequired 1  
        AmountIncrement 1  
        NumberOfModifications 4  
    }  
    Instances 1  
}  
  
Source {  
    CallGenerator {  
        Resources Galileo.Library.Connection.SimpleConnectionResources  
        Component Galileo.Library.Source.DeterministicCallGenerator  
        FirstConnectionDelay 10  
        InterArrivalDelay 30  
        ConnectionDuration 180  
        ConstellationRestricted true  
        Source 0  
        Destination 1  
        ResourcesAmountRequired 1  
        AmountIncrement 1  
        NumberOfModifications 0  
    }  
    Instances 2  
}
```

Figure 3.5: A sample configuration file for the source segment.

3.2.4 The source segment file

Figure 3.5 shows a sample configuration file for the source segment.

The configuration is made of one or more **Source** blocks. Each block defines a set of sources sharing the same call generator, hence being statistically dependent. A source definition is made of:

- A **CallGenerator** block specifying the type of connection resources the call generator handles (**ConnectionResources** keyword), the component implementing the call generator (**Component** keyword) and parameters depending on the call generator type.
- An **Instances** keyword specifying how many sources using this call generator must be instantiated.

In the sample file of figure 3.5 three sources are created. The last two sources share the same call generator.

Since a call generator specifies which are the source and destination stations of a connection, a call generator must be aware of the number of stations existing as well as their location. This relation does not appear in the configuration files and is implicitly coded in the call generator.

3.3 Running

Depending on whether GaliLEO was compiled with a JDK or gcj, either `make sungo` or `make gcjgo` are used to run GaliLEO. Both specifies as configuration file (*experiment* which is the root for the filenames of the four configuration files.

GaliLEO displays then a welcome message and starts by parsing the configuration files. GaliLEO does not have (yet) a graphical user interface. There is no other mean for stopping GaliLEO than the infamous `Ctrl-C` key combination.

3.4 Getting the results

The results are logged either on screen or in files depending on the configuration. In the case of output to files, each measure corresponds to a file whose name is based on the measure name.

3.5 A sample experiment

GaliLEO is shipped with a set of configuration file (and related components) implementing a simple experiment. This experiment was used during the debugging phase of GaliLEO.

3.5.1 The constellation

3.5.2 The traffic model

3.5.3 The stations

3.5.4 The satellite on-board processing

Chapter 4

The Simulation Engine

4.1 Discrete event simulation

4.2 It's all about components

4.2.1 Code components

4.2.2 Customisable code components

4.2.3 Data components

4.2.4 Customisable code components

4.3 Being creative

4.3.1 Creating a new agenda

4.3.2 Creating a new scheduler

4.3.3 Creating new components

DRAFT

Chapter 5

The Components

DRAFT

Chapter 6

Examples of customisation

- 6.1 Creating new traffic sources**
- 6.2 Creating a new ISL routing algorithm**
- 6.3 Creating a new link state manager**
- 6.4 Creating a new connection change manager**
- 6.5 Creating new measures**